Лабораторна робота №4

Програмування арифметичних операцій підвищеної розрядності

Мета: Навчитися програмувати на асемблері основні арифметичні операції підвищеної розрядності, а також отримати перші навички програмування власних процедур у модульному проекті.

Завдання:

1. Створити у середовищі MS Visual Studio проект з ім’ям **Lab4**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути три модуля на асемблері:
   - головний модуль: файл **main4.asm**. Цей модуль створити та написати заново, частково використавши текст модуля main3.asm попередньої роботи №3;
   - другий модуль: використати **module** попередньої роботи №3;
   - третій модуль: створити новий з ім'ям **longop**.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуємий файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – кодовані значення чисел згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати, вихідний текст та дизасемблерний машинний код програми.

Теоретичні відомості

Поняття "підвищена розрядність" тут означає таку кількість біт, яку процесор не може обробити одною командою. Наприклад, у процесорах 32-бітової архітектури арифметичні команди виконуються для 32-бітових чисел, і таку саму розрядність мають робочі регістри (EAX, EBX, ECX, EDX тощо). Процесор 64-бітовий здатен одною командою додавати вже 64-бітові числа. Проте, може виникнути потреба у роботі з числовими даними, розрядність яких може суттєво – у десятки разів перевищувати довжину машинного слова. Підвищена розрядність означає підвищену точність та (або) більший діапазон числових даних.

Реалізувати арифметику підвищеної розрядності можна власноруч програмним шляхом. Мова асемблера дозволяє це реалізувати достатньо просто та ефективно.

Додавання

Нехай потрібно додати 128-бітові числа. Представимо кожне 128-бітове число чотирма 32-бітовими групами: A_i та B_i . Спочатку треба додати молодші групи бітів: A_0+B_0 . Результат додавання буде 33-бітовим – 32 молодші біти та перенос. Молодші 32 біти вже є бітами результату, а перенос треба додати при обчисленні суми наступних груп: A_1+B_1+перенос. І так далі. Біт переносу результату додавання старших груп A_3+B_3 потрібно зберегти, якщо у цьому є потреба. Алгоритм додавання наведений на рис. 1.

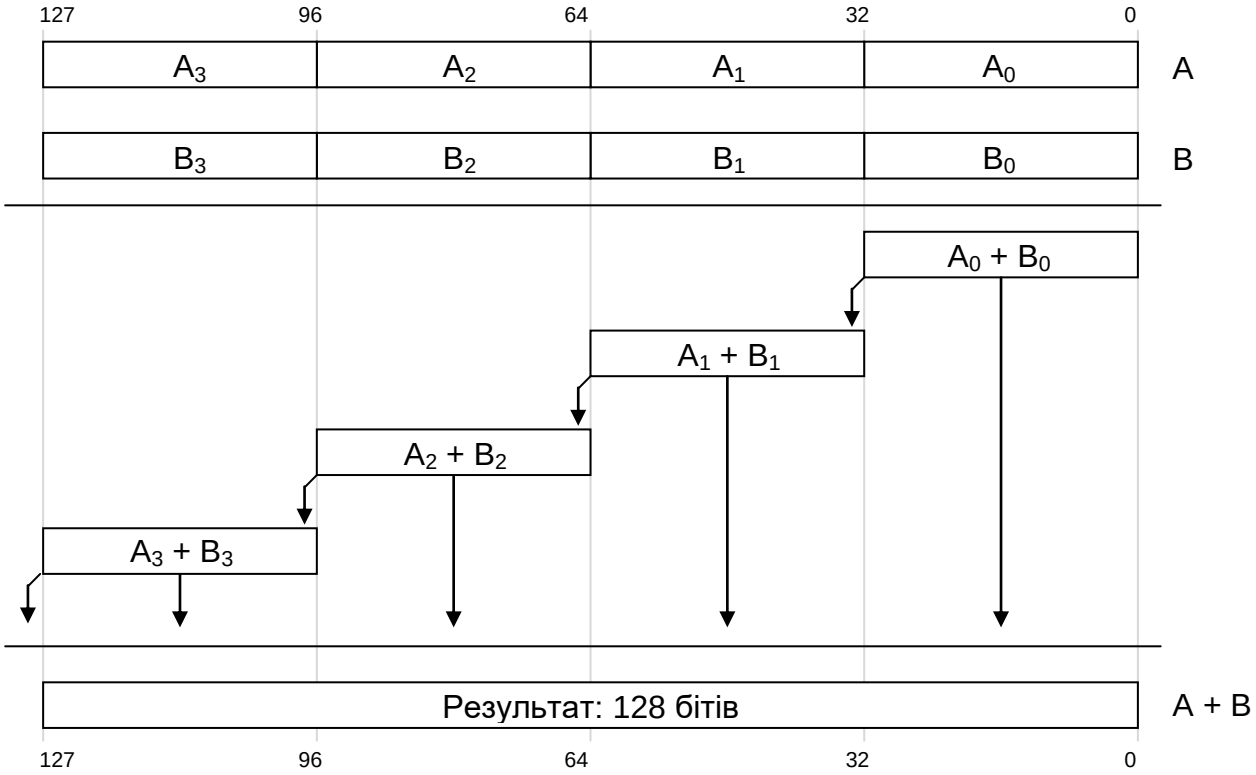

Рис. 1. Алгоритм додавання 128-бітових чисел

Такий алгоритм коректно працює як з числами без знаку, так і з числами зі знаком у додатковому коді.

При реалізації додавання у програмах на асемблері використовується команда ADD для додавання молодших груп бітів A_0+B_0, і команда ADC – для додавання у вигляді: A_i+B_i+перенос. При виконанні додавання значення переносу автоматично записується у біт CF регістру EFLAGS.

Віднімання

Віднімання виконується так само, як і додавання – послідовно від молодших груп бітів. При відніманні наступних груп бітів потрібно враховувати результат віднімання попередньої групи.
Розглянемо віднімання на прикладі. Нехай потрібно від нуля відняти одиницю:

```
00..00 00..00 00..00 00..00   A
00..00 00..00 00..00 00..01   B
---------------------------
                     11..11   A0-B0
               11..11         A1-B1-позичання
       11..11                 A2-B2-позичання
11..11                        A3-B3-позичання
---------------------------
11..11 11..11 11..11 11..11   Результат A-B
```

Результат віднімання записується у додатковому коді. У розглянутому вище прикладі це додатковий код числа (-1).

Позичання виникає при відніманні більшого числа від меншого. Позичання означає вимогу розповсюдити групу одиниць від старшого до поточного біту. Цю групу одиниць достатньо закодувати одним бітом, який зветься бітом позичання (borrow).

При реалізації у програмах на асемблері для віднімання молодших груп бітів ($A_0 – B_0$) використовується команда SUB, а віднімання усіх наступних груп бітів ($A_i – B_i$) робиться командою SBB (subtract with borrow). Біт позичання записується у біт CF регістру EFLAGS.

Щодо програмної реалізації на асемблері додавання та віднімання

Програмний код можна оформити у вигляді процедур, які викликаються для виконання відповідної операції. Рекомендується надати процедурам імена, які позначають їхнє функціональне призначення, а також приналежність певному модулю, наприклад:

Add_128_LONGOP – процедура додавання 128-бітових даних;
Sub_128_LONGOP – процедура віднімання 128-бітових даних;

Кожна з таких процедур буде мати три параметри: операнд A, операнд B, та результат. При розробці будь-якої процедури, потрібно спочатку узгодити, що і як передавати процедурі у якості параметрів і у якому вигляді і як отримувати результат. У якості, наприклад, 128-бітових операндів та результату можна визначити масиви з чотирьох 32-бітових подвійних слів, прямо ініціалізувавши їх деякими числовими значеннями:

```
ValueA dd 80010001h, 80020001h, 80030001h, 80040001h
ValueB dd 80000001h, 80000001h, 80000001h, 80000001h
Result dd 4 dup(0)
```

Для процедур такі параметри зручно передавати у вигляді вказівників, адрес цих масивів. Наприклад:

```
push offset ValueA
push offset ValueB
push offset Result
call Add_128_LONGOP
```

Таким чином, три параметри процедурі передаються через стек. Процедура повинна прочитати значення з масивів ValueA, ValueB і записати результат у масив Result по вказаній адресі. Процедура сама повинна відновити стек. Такий програмний код буде для модуля-користувача процедури **Add_128_LONGOP.** Так само можна зробити для процедури віднімання.

Після такого зовнішнього визначення інтерфейсу обміну інформацією із процедурами, тепер можна переходити до програмування внутрішньої реалізації самих процедур. Фрагмент можливого варіанту реалізації процедури додавання:

```
; результат = A + B
Add_128_LONGOP proc
   push ebp
   mov ebp,esp

   mov esi, [ebp+16]          ;ESI = адреса A
   mov ebx, [ebp+12]          ;EBX = адреса B
   mov edi, [ebp+8]           ;EDI = адреса результату

   mov eax, dword ptr[esi]    ;починаємо з молодших 32-бітових груп
   add eax, dword ptr[ebx]    ;додавання 32-біт
   mov dword ptr[edi], eax    ;запис молодших 32 біт суми

   mov eax, dword ptr[esi+4]  ;наступна 32-бітова група
   adc eax, dword ptr[ebx+4]  ;додавання з урахуванням переносу з молодшої
   mov dword ptr[edi+4], eax  ;запис наступних 32 біт суми
```

```
   mov eax, dword ptr[esi+8]    ;наступна 32-бітова група
   adc eax, dword ptr[ebx+8]    ;додавання з урахуванням переносу з молодшої
   mov dword ptr[edi+8], eax    ;запис наступних 32 біт суми

   . . .                        ;і так далі

   pop ebp                      ;відновлення стеку
   ret 12
Add_128_LONGOP endp
```

Додавання багаторозрядних чисел тут запрограмовано послідовним додаванням 32-бітових груп, починаючи з групи молодших розрядів. Для додавання наймолодших розрядів виконується команда ADD. В результаті може виникнути перенос у наступну старшу групу. Цей перенос записується у біт CF регістру EFLAGS. Для додавання наступної 32-бітової групи бітів з урахуванням можливого переносу використана команда ADC. І так далі, до найстаршої 32-бітової групи.

Команда RET означає завершення роботи процедури. Тут ця команда також звільнює стек від 3 чотирибайтових параметрів.

У подібний спосіб можна запрограмувати процедуру **Sub_128_LONGOP,** призначену для віднімання 128-бітних даних. Обробку виконувати також 32-бітними групами, починаючи з молодших. Віднімання наймолодших 32 біт робити командою SUB, а наступні 32-бітові групи обробляти командою SBB.

Можна звернути увагу на те, що у вихідному тексті процедури додавання декілька разів повторюється майже однаковий код з команд MOV та ADC:

```
mov eax, dword ptr[esi+Зсув]    ;наступна 32-бітова група
adc eax, dword ptr[ebx+Зсув]    ;додавання з урахуванням переносу з молодшої
mov dword ptr[edi+Зсув], eax    ;запис наступних 32 біт суми
```

Значення зсуву – 0, 4, 8, і так далі, відповідно розрядності операції. Для зменшення обсягу вихідного тексту можна запрограмувати циклічне повторення цих команд з відповідним змінним значенням зсуву. При програмуванні циклу, потрібно враховувати необхідність зберігання біту переносу CF – або якось зберігати цей біт, або для організації циклу використовувати команди, які б не змінювали біт переносу.

Програмування циклів

Цикл є дуже популярним елементом різноманітних алгоритмів. Як запрограмувати цикл на асемблері?

Нехай потрібно щось виконати 10 разів. Це "щось" будемо називати "тілом циклу". Тіло циклу може містити деяку множину рядків програмного коду на асемблері. Розглянемо варіант алгоритму циклу з постумовою (рис. 2)

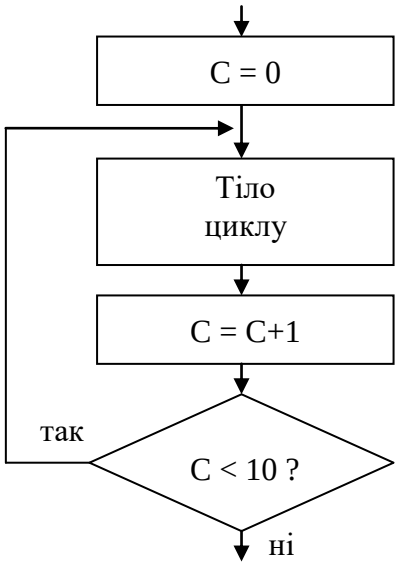

Рис. 2. Блок-схема алгоритму циклу з постумовою

На асемблері такий цикл можна запрограмувати наступним чином:

```
  mov ecx, 0        ; обнулюємо лічильник – регістр ECX
cycle:
  . . .             ; тіло циклу

  inc ecx           ; збільшуємо лічильник на 1
  cmp ecx, 10       ; порівнюємо лічильник з 10
  jl cycle          ; якщо лічильник менше – перехід на мітку cycle
```

Необхідно відзначити, що такий варіант організації циклу є неприйнятний для послідовної обробки груп бітів з розповсюдженням переносу, оскільки команда CMP сама записує біт CF – тобто знищує перенос, який треба було б розповсюджувати.

Інший варіант циклу – у лічильник спочатку записується потрібна кількість повторень, а на кожній ітерації значення лічильника зменшується на одиницю:

```
  mov ecx, 10       ; у регістр ECX записуємо кількість повторень
cycle:
  . . .             ; тіло циклу

  dec ecx           ; зменшуємо лічильник на 1
  jnz cycle         ; якщо лічильник не 0, то перехід на мітку cycle
```

Такий варіант циклу може бути використаний для наших цілей, оскільки ані команда DEC, ані JNZ не змінюють біт CF регістру EFLAGS.

Як вже вказувалося вище, при додаванні чисел підвищеної розрядності виконується обробка 32-бітових за такою схемою: спочатку командою ADD додаються молодші 32 біти, а решта груп бітів додаються командою ADC. Для того, щоб запрограмувати це у циклі, команди тіла циклу повинні бути однаковими. Тіло циклу можливо побудувати на основі команд ADC, якщо попередньо обнулити біт переносу командною CLC:

```
  mov ecx, ...     ; ECX = потрібна кількість повторень
  clc              ; обнулює біт CF регістру EFLAGS
cycle:
  mov eax, dword ptr[esi+Зсув]
  adc eax, dword ptr[ebx+Зсув]       ; додавання групи з 32 бітів
  mov dword ptr[edi+Зсув], eax
  . . .            ; модифікація зсуву
  dec ecx          ; лічильник зменшуємо на 1
  jnz cycle
```

Для кожної ітерації потрібно сформувати адресу поточної групи бітів, наприклад, адреса

```
dword ptr[esi+Зсув]
```

складається з вмісту регістру ESI та зсуву. Зсув можна зберігати у якомусь регістрі, наприклад, EDX і при переході до наступної ітерації збільшувати його на одиницю:

```
  mov edx, 0
  . . .
cycle:
  mov eax, dword ptr[esi+4*edx]
  . . .
  inc edx
  dec ecx
  jnz cycle
```

Необхідно відзначити, що наведені тут відомості не вичерпують усіх можливостей вирішення подібних завдань.

Порядок виконання роботи та методичні рекомендації

1. Створіть у середовищі MS Visual Studio новий проект з ім’ям **Lab4**.

2. Додайте у проект порожній файл з ім’ям **main4.asm**. Цей файл буде головним файлом програмного коду. Для спрощення виконання роботи скористайтеся текстом головного файлу *.asm попередньої роботи №3. Скопіюйте текст і у

вікні редагування вихідного тексту вилучіть зайві рядки. Запишіть на диск головний файл програми **main4.asm**.

3. Додайте у проект модуль з ім'ям **module**. У проекті використовується файли **module.asm, module.inc** попередньої роботи №3 без будь-яких змін. Рекомендація: для того, щоб у декількох проектах використовувати ті самі модулі, запишіть файли цих модулів у окрему папку. Кожний файл вихідного тексту модулів, які спільно використовується, повинен бути у одному екземплярі.

4. Додайте у проект новий модуль з ім'ям **longop**. Файл вихідного тексту **longop.asm** буде містити вихідний текст процедур, які реалізують арифметичні операції підвищеної розрядності. Окрім файлу **longop.asm** потрібно створити файл заголовку **longop.inc**. Цей файл міститиме імена процедур, які будуть викликатися з інших модулів. Приклад вмісту файлу **longop.inc**:

```
EXTERN Add_128_LONGOP : proc
EXTERN Sub_128_LONGOP : proc
```

5.Програмування числових значень операндів із розрядністю згідно завданню. У файлі **main4.asm** потрібно запрограмувати створення перемінних потрібної розрядності, ініціалізувавши їх числовими значеннями згідно варіанту завдання.

6. Запрограмувати вивід результатів виконання операцій для кожного набору значень у окремому діалоговому вікні MessageBox.

7. Компіляція, виклик програми, налагодження, отримання результатів. Виконання цих дій виконується у середовищі MS Visual Studio. Відомості та методичні рекомендації надані у відповідних розділах попередніх робіт №2, 3.

Зміст звіту:

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів, вихідного тексту та дизасемблерного машинного коду
6. Висновки

Варіанти завдання

Номер варіанту (N) згідно списку студентів у журналі.

Кожному студенту необхідно запрограмувати на асемблері два додавання (A+B) та одне віднімання (A-B).

1. Числові значення операндів A та B

Операнди A та B для першого додавання наступні:

```
A = 80010001h, 80020001h, 80030001h, 80040001h, ...
B = 80000001h, 80000001h, 80000001h, 80000001h, ...
```
(у шістнадцятковому коді)

Операнди A та B для другого додавання:

```
A = (N 32-біт), (N+1 32-біт), (N+2 32-біт), (N+3 32-біт), ...
B = 80000001h, 80000001h, 80000001h, 80000001h, ...
```
де N – номер варіанту

Операнди для віднімання такі:

```
A = 0
B = (N 32-біт), (N+1 32-біт), (N+2 32-біт), (N+3 32-біт), ...
```
де N – номер варіанту

Числові значення операндів тут вказані від молодших ліворуч до старших праворуч розрядів групами по 32 біт. Записані явно перші 128 біт, далі відповідно варіанту розрядності.

2. Як формувати числові значення операндів перед виконанням операцій

Для номерів варіантів N = 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, . . .
операнди для **першого додавання** формувати у циклі, решту – без циклу

Для номерів варіантів N = 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, . . .
операнди для **другого додавання** формувати у циклі, решту – без циклу

Для номерів варіантів N = 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, . . .
операнди для **віднімання** формувати у циклі, решту – без циклу

3. Розрядність операндів A та B для кожного варіанту завдання

№ варіанту	Розрядність додавання (біт)	Розрядність віднімання (біт)
1	96	1024
2	128	992
3	160	960
4	192	928
5	224	896
6	256	864
7	288	832
8	320	800
9	352	768
10	384	736
11	416	704
12	448	672
13	480	640
14	512	608
15	544	576
16	576	544
17	608	512
18	640	480
19	672	448
20	704	416
21	736	384
22	768	352
23	800	320
24	832	288
25	864	256
26	896	224
27	928	192
28	960	160
29	992	128
30	1024	96

Контрольні питання:

1. Як виконується додавання підвищеної розрядності?
2. Як виконується віднімання підвищеної розрядності?
3. Як передаються параметри процедурам?
4. Якими командами виконується додавання та віднімання?
5. Що таке перенос і куди він записується?
6. Як запрограмувати цикл на асемблері?
7. Що виконують команди DEC, INC, JNZ, CLC?